LAIO: Lazy Asynchronous I/O For Event-Driven Servers

Khaled Elmeeleye

Thesis: Master of Science
Computer Science
Rice University, Houston, Texas (December 2003)
RICE UNIVERSITY

LAIO: Lazy Asynchronous I/O For Event-Driven Servers

by

Khaled Elmeleegy

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Alan L. Cox, Chair
Associate Professor of Computer Science

Dr. Willy Zwaenepoel
Dean
Faculty of Computer & Communications Sciences
Swiss Federal Institute of Technology

Dr. David B. Johnson
Associate Professor of Computer Science

HOUSTON, TEXAS

DECEMBER, 2003
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
LAIO: Lazy Asynchronous I/O For Event-Driven Servers

Khaled Elmeleegy

Abstract

In this thesis, I introduce Lazy Asynchronous I/O (LAIO), a new API for performing I/O that is well-suited but not limited to the needs of high-performance, event-driven servers. In addition, I describe and evaluate an implementation of LAIO that demonstrably addresses certain critical limitations of the asynchronous and non-blocking I/O support in present Unix-like systems. LAIO is implemented entirely at user-level, without modification to the operating system’s kernel. It utilizes scheduler activations. Using a micro-benchmark, LAIO was shown to be more than 3 times faster than AIO when the data was already available in memory. It also had a comparable performance to AIO when actual I/O needed to be made. An event driven web server (thttpd) achieved more than 38% increase in its throughput using LAIO. The Flash web server’s throughput, originally achieved with kernel modifications, was matched using LAIO without making kernel modifications.
Acknowledgments

I thank my advisors Dr. Alan L. Cox and Dr. Willy Zwaenepoel for the time and effort they invested in me and my work. Special thanks to Alan for his help, ideas and suggestions that were crucial to get this work done. Special thanks to Willy for his guidance and providing positive feedback and following my work from the other side of the Atlantic at Lausanne, Switzerland.

Many thanks to Dr. David Johnson for his valuable suggestions regarding my thesis. I thank my colleague, Anupam Chanda for his help and suggestions. Finally, I thank my family and friends for their support and encouragement.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi
List of Tables vii

1 Introduction 1

2 LAIO API 5
2.1 Interface ................................. 5
2.2 An LAIO example ...................... 6

3 LAIO Implementation 8

4 Event Driven Programming with Various I/O API 11
4.1 Event driven programming with LAIO ............... 12
4.2 Event driven programming with non-blocking I/O ........ 13
4.3 Event driven programming with AIO .................. 13

5 Servers Studied 17
5.1 thttpd .................................. 17
5.2 Flash .................................. 17

6 Evaluation Methodology 21

7 Results 23
7.1 Microbenchmarks: LAIO vs. non-blocking I/O vs. POSIX AIO 23
7.2 thttpd Results 24
7.3 Flash Results 28
  7.3.1 mmap Results 29
  7.3.2 sendfile Results 31

8 Related Work 34

9 Conclusions 35

Bibliography 36
Illustrations

2.1 An Example of LAIO ........................................ 7

3.1 LAIO Syscall ........................................... 8

4.1 Pseudo code of an event loop .......................... 11
4.2 An Example of an Event Handler Performing LAIO ... 12
4.3 An Example of an Event Handler Performing Non-Blocking I/O ... 15
4.4 An Example of Asynchronous I/O .................... 16

7.1 Throughput for Berkeley workload with thttpd ........ 25
7.2 Response time for Berkeley workload with thttpd .... 25
7.3 Throughput for Rice workload with thttpd .......... 26
7.4 Response time for Rice workload with thttpd ....... 26
7.5 Throughput for Rice workload with thttpd with reduced memory size ... 27
7.6 Response time for Rice workload with thttpd with reduced memory size ... 28
Tables

6.1 Web Trace Characteristics .................................................. 21

7.1 Mmap results: Throughput for the Rice workload with 500 clients . . . . 29
7.2 Mmap results: Throughput for the Berkeley trace with 500 clients . . . . 29
7.3 Sendfile results: Throughput for the Rice workload with 500 clients . . . . 32
7.4 Sendfile results: Throughput for the Berkeley trace with 500 clients . . . . 32
Chapter 1

Introduction

In this thesis, I introduce *Lazy Asynchronous I/O* (LAIO), a new API for performing I/O that is well-suited but not limited to the needs of high-performance, event-driven servers. In addition, I describe and evaluate an implementation of LAIO that demonstrably addresses certain critical limitations of the asynchronous and non-blocking I/O support in present Unix-like systems.

In general, to achieve the highest level of performance, an event-driven server must avoid blocking on any operation, from I/O to resource allocation. Thus, in event-driven servers, the use of asynchronous or non-blocking I/O, is for all practical purposes, mandatory. The problem is that the asynchronous and non-blocking I/O support in present Unix-like systems is limited by its lack of general applicability. For example, non-blocking I/O can be performed on network connections, but not disk files. In contrast, POSIX asynchronous I/O (AIO) can be performed on disk files, but only supports reading and writing; many widely-used operations that require disk access as a part of their implementation, such as opening a file or determining its size, are not supported.

In principle, this problem could be addressed by changes to the operating system. Such changes would affect the operating system’s interface as well as its implementation. In practice, the scope of such changes has effectively impeded such a solution. As a consequence, developers faced with this problem have either (1) abandoned an event-driven architecture entirely for a multithreaded or multiprocess architecture, (2) accepted that some operations can block and the effect thereof on performance, or (3) simulated asynchronous
I/O at user-level by submitting blocking operations to a queue that is serviced by a pool of threads.

The first of these options has received considerable attention in the literature. The consensus conclusion being that an event-driven architecture has potential performance advantages over a multithreaded or a multiprocess architecture [4–6, 13]. These advantages include greater control over scheduling, lower overhead for maintaining state, and lower overhead for synchronization. Recently, von Behren et al. [12] have, however, argued that a better designed threading library can enable the multithreaded architecture to achieve performance comparable to the event-driven architecture. For my part, I only observe that performance is not a reason to switch. Thus, I do not give this option further consideration.

Surprisingly, the second option does appear in practice. The httpd web server is a notable example.

The asymmetric multiprocess event driven (AMPED) architecture that was employed by the Flash web server is representative of the third category [6]. The essence of its solution was a hybrid architecture that consisted of an event-driven core augmented by helper threads. Flash performs all non-blocking operations in an event-driven fashion and all potentially blocking operations are dispatched to helper threads.

The LAIO interface consists of three simple, easy to use functions. The essence of LAIO is to perform asynchronous I/O lazily. If the I/O operation completes without blocking then to the program it looks exactly like an ordinary synchronous I/O operation. In fact, there is no significant overhead. If, however, the operation would block, the program receives an indication, EINPROGRESS, that the operation is being performed asynchronously. On completion of the operation, a notification is delivered to the program.

LAIO is implemented entirely at user-level, without modification to the operating system’s kernel. It only requires support for scheduler activations [2]. I use scheduler acti-
vations to deliver upcalls to the LAIO library when a process blocks and later unblocks in the kernel. In effect, these upcalls provide a method for converting arbitrary, potentially blocking I/O operations into asynchronous operations. Coupled with stack switching, these upcalls allow us to continue the execution of the application even if the process blocks. Stack switching is achieved by saving the context of the executing thread before it blocks, and jumping to the saved context on receiving an upcall when the thread blocks.

The contributions of this thesis are three-fold.

First, the LAIO API is simpler to use than non-blocking I/O, and I illustrate this in the thesis. To demonstrate the usability of LAIO I have integrated it with libevent, an event notification library [8]. I have also augmented the thttpd web server [7], which is an event driven web server, and the Flash web server [6] to use the LAIO API.

Second, the cost of LAIO is small compared to non-blocking I/O or POSIX asynchronous I/O. I constructed microbenchmarks and found that LAIO is marginally more expensive than non-blocking I/O. Compared to POSIX asynchronous I/O, LAIO is three times faster when the operation runs to completion without blocking. Performance is comparable when the operation blocks and has to be performed asynchronously.

Third, LAIO achieves better or comparable performance when integrated with existing event-driven servers. Using LAIO, thttpd attained 38% more throughput. Flash integrated with LAIO matched the performance of Flash which used kernel modifications to prevent blocking of the server.

The remainder of this thesis is organized as follows. Chapter 2 describes the LAIO API and provides an example of using LAIO. Chapter 3 describes the LAIO implementation. Chapter 4 describes how to use different I/O APIs in event driven programming and explains how LAIO is better than previous APIs. Chapter 5 describes my modifications to the thttpd and Flash web servers. Chapter 6 describes my evaluation methodology. Chapter 7
describes the experiments done. Then I discuss the related work in Chapter 8. Finally, Chapter 9 concludes this thesis.
Chapter 2

LAIO API

LAIO is an I/O library that allows applications to issue I/O operations asynchronously. Although it is mainly intended to be used with I/O, it can also be used to convert any blocking system call into an asynchronous one. In this chapter, I first present the LAIO API and then describe its usage walking through an example.

2.1 Interface

The API for LAIO consists of three functions: laio_syscall(), laio_gethandle(), and laio_poll().

laio_syscall() has the same signature as syscall(), a standard function for performing indirect system calls. The first parameter identifies the desired system call. Symbolic names for this parameter, representing all system calls, are defined in a standard header file. The rest of the parameters vary according to the expectations of the desired system call. If the desired system call is able to complete without blocking, the behavior of laio_syscall() is indistinguishable from that of syscall(). If, however, the system call is unable to complete without blocking, laio_syscall() returns -1, setting the global variable errno to EINPROGRESS. Henceforth, I refer to this case as "a background laio_syscall()."

laio_gethandle() returns an opaque handle for the purpose of identifying a background laio_syscall(). Specifically, this handle identifies the most re-
cent laio_syscall() that reported EINPROGRESS. If, however, the most recent laio_syscall() completed without blocking, laio_gethandle() returns NULL. In other words, laio_gethandle() is expected to appear shortly after a background laio_syscall().

laio_poll() returns a set of structures. Each structure represents the completion of a background laio_syscall(). A structure consists of a handle, a return value, and a possible error code. The handle identifies a background laio_syscall(). The return value and possible error code are determined by the particular system call that was performed by the background laio_syscall().

2.2 An LAIO example

I present an example of writing data to a network socket using LAIO. There are three steps to performing an I/O. Figure 2.1 illustrates these steps.

First, the write operation is initiated using the laio_syscall(). The operation may or may not block. If it doesn’t block, number of bytes written is returned and execution continues normally. However if it blocks a return value of -1 is returned with errno set to EINPROGRESS. At this point, the application can engage in other activities. The application uses polling to determine if the operation has finished. The operation is finished if laio_poll() returns a set of ready laio handles containing the handle of the write that was just issued.

*In a multithreaded environment, the intended meaning of “most recent laio_syscall()” is the most recent laio_syscall() that was performed by the same thread.
/* Step 1: attempt the operation */
return_value = laio_syscall(SYS_write,
    client->socket,
    client->buffer,
    client->bytes_to_write);

if (return_value == -1) {
    if (errno == EINPROGRESS) {
        /* Step 2: perform other activities until
         * the completion of the LAIO */

        handle = laio_gethandle();

        for (;;) {
            ...
            /* perform other activities */
            ...
        /* Step 3: poll for completion */
        if (laio_poll(fds, nfds, timeout) < 0)
            /* handle fatal error */
        else
            for (i = 0; i < nfds; i++)
                if (fds[i].laio_desc == handle)
                    return_value = fds[i].laio_rv;
            return_value = fds[i].laio_rv;
            goto done;

        }
    }
    else {
        /* handle fatal error */
    }
}

done:
/* operation completed successfully */

Figure 2.1: An Example of LAIO
Chapter 3

LAIO Implementation

I have implemented the LAIO API as a user-level library. Now I will explain the operation and the implementation of the LAIO APIs presented in chapter 2.

Figure 3.1: LAIO Syscall

Figure 3.1 illustrates the operation of `laio.syscall()`. It is a wrapper around any system call, where it saves the current thread's context and enables upcalls to be delivered. It then invokes the system call. If the system call doesn't block, it returns immediately to the application with the corresponding return value and upcalls are disabled. However if the
system call blocks, blocking the current thread in the kernel, an upcall is generated by the kernel invoking the upcall handler on a new thread. The upcall handler then steals the previous thread's stack using the context previously saved by laio.syscall(). Now running on the previous threads stack, the upcall handler returns from laio.syscall() with the return value set to -1 and the errno set to EINPROGRESS. The EINPROGRESS errno value notifies the application that the laio.syscall() has blocked in the kernel and a background laio.syscall() is in progress. It then makes a call to laio.gethandle() to get the LAIO descriptor associated with the last asynchronous I/O. It associates a continuation function with that LAIO descriptor. This continuation function is invoked after the background laio.syscall() completes.

Unblocking of the background laio.syscall() generates another upcall. This upcall returns the LAIO descriptor for the laio.syscall() that had earlier blocked. The library adds this descriptor to a list of descriptors corresponding to background laio.syscall()s that have completed. The application calls laio.poll() to retrieve this list. The application then invokes the continuation function associated with each descriptor from this list. One thing to note here is that when the background laio.syscall() completes it does not resume execution from the point where it blocked. Instead, a continuation function associated with the background laio.syscall() is invoked.

The LAIO library maintains a pool of LAIO descriptors. It associates a descriptor with each laio.syscall(). If a laio.syscall() blocks, the upcall returns the corresponding LAIO descriptor. This is followed by a call to laio.gethandle(), which returns the LAIO descriptor corresponding to the background laio.syscall().

I require scheduler activations [2] to provide upcalls for the implementations of the LAIO library. Currently, many operating systems support scheduler activations, including
FreeBSD [10], NetBSD [14], Solaris, and Tru64. They provide a way of communication between the kernel and the user level program about events the application thread goes through inside the kernel. Blocking of the thread in the kernel, due to I/O, for example, and later unblocking of the thread are principal examples of such events. Such information could be used at the user level to make scheduling decisions. This information is provided to the application via upcalls. When the application thread blocks a new thread is spawned by the kernel and the upcall is provided on this thread. Only one upcall is provided for each blocking or unblocking event. When a thread unblocks, it does not start running spontaneously. Instead an upcall is delivered to the currently executing thread of the application. The application thread then resumes the unblocked thread. This means on a multi-processor system, a thread can not start or unblock before the upcall is delivered. And each upcall is delivered to a single processor eliminating any race conditions between threads.
Chapter 4

Event Driven Programming with Various I/O API

In this chapter, I first provide some background about event driven programming. Then I describe the usage of different I/O APIs with event driven programming, namely Lazy Asynchronous I/O (LAIO), non-blocking I/O and asynchronous I/O (AIO) [11].

An event driven server has an event loop which is an infinite loop that receives event notifications, for example, a socket ready for read or write. For each of those received events, the event loop dispatches the corresponding event handler. Event handlers run to completion and are never preempted. Consequently it is sufficient to guarantee that the shared state is consistent before returning to the event loop. Figure 4.1 shows pseudo code of an event loop.

```plaintext
for (;;) {
    /* Step 1: poll for ready events,
    * receive a list of events (evs) */
    evs = poll(...) 
    ...
    /* perform other activities */
    ...
    /* Step 2: for each event in evs
    * dispatch its handler */
    for each event (e) in (evs)
        handle (e , ...)
}
```

Figure 4.1: Pseudo code of an event loop
4.1 Event driven programming with LAIO

Figure 4.2 illustrates an event handler using LAIO. Specifically, this event handler performs a write() using laio_syscall(). If the laio_syscall() reports EINPROGRESS, the function client_response_epilogue() is registered to be invoked when the write operation has completed.

When LAIO is used for building the event driven server, laio_poll() should be used for polling at the event loop of Figure 4.1.

```c
/* Step 1: attempt the operation */
return_value = laio_syscall(SYS_write,
                         client->socket,
                         client->buffer,
                         client->bytes_to_write);

if (return_value == -1) {
    if (errno == EINPROGRESS) {
        /* Step 2: tell the event loop to call
        * client_response_epilogue upon
        * completion of the LAIO */
        event_set(&client->event, laio_gethandle(),
                  EV_LAIO_COMPLETED,
                  client_response_epilogue, client);
        event_add(&client->event, NULL);
    } else {
        /* handle fatal error */
    }
    return; /* to the event loop */
}
// completed without blocking */
client_response_epilogue(..., client);
return; /* to the event loop */
```

Figure 4.2: An Example of an Event Handler Performing LAIO

In order to demonstrate the flexibility and completeness of the LAIO API, I augmented Niels Provos' libevent [8], which is a general-purpose event notification library, with support for LAIO. Libevent has been used to implement numerous event-driven servers.
4.2 Event driven programming with non-blocking I/O

Figure 4.3 illustrates the same event handler presented in Figure 4.2 but using non-blocking I/O instead of LAIO.

In contrast to Figure 4.2, the application is required to maintain state on the progress of the I/O operation which is a shortcoming of non-blocking I/O. Also, unlike LAIO, it may be required for the write() to be invoked more than once for the write operation to complete. Another shortcoming of non-blocking I/O is that it is limited to sockets, pipes, and terminals but not files. For non-blocking I/O, polling in the event loop could be done using any standard event notification mechanism like select(), poll(), or kevent().

4.3 Event driven programming with AIO

In this section I first introduce AIO. Then I discuss how to use AIO in an event handler. Using AIO there are four steps to performing an I/O. Figure 4.4 illustrates these steps. First, a control block describing the operation is initialized. This control block includes information such as the descriptor on which the operation is performed, the location of the buffer, and its size. Then, the operation is initiated. At this point, the application can engage in other activities. The control block is still, however, used as a handle to identify the unfinished operation. In most implementations, the application determines that the operation has finished through polling or an asynchronous event notification mechanism, such as signals. For the sake of simplicity, Figure 4.4 illustrates polling. The single operation, aio_error(), is used to both poll for completion and to learn of any error that occurred during the operation. Finally, the return value of the operation is obtained, using aio_return().
From Figure 4.4 we see that at least three calls are required for the write operation to be completed. This happens even if the operation doesn't block. LAIO does not have this overhead. Similar to non-blocking I/O, the POSIX AIO API has a major limitation which prevents it from being an universal I/O API. This limitation is the small set of supported operations. Only the basic operations, reading and writing, are supported. In other words, complex operations that include I/O as part of their implementation, such as opening a file or determining its size, are not supported.

An event handler using AIO will be very similar to that in Figure 4.2. The only difference, other than using the AIO API instead of the LAIO API, is that the write operation is handled asynchronously even if it does not block. With AIO, the event loop can use event notification mechanisms like kevent().
void
client_response_write(..., void *arg)
{
  struct client *client = arg;
  /* Step 0: assume that the one-time operations,
  * enabling non-blocking I/O and initializing
  * the state of progress, have been performed elsewhere. */
  ...
  /* Step 1: attempt the operation */
  return_value = write(client->socket,
                        &client->buffer[client->bytes_written],
                        client->bytes_remaining);
  if (return_value == client->bytes_remaining) {
    /* tell the event loop that the operation has completed */
    event_del(&client->event);
    client_response_epilogue(client, ...);
    return; /* to the event loop */
  } else if (return_value > 0) {
    /* and implicitly less than bytes_remaining */

    /* Step 2: update the state of progress */
    client->bytes_written += return_value;
    client->bytes_remaining -= return_value;
    else if (return_value == -1 && errno != EAGAIN) {
      /* handle fatal error;
      * tell the event loop not to continue */
      event_del(&client->event);
      client_response_error(client, ...);
      return; /* to the event loop */
    }
    /* assume that this function remains registered
    * with the event loop so that it will be called
    * when the socket is again ready for writing */
    return; /* to the event loop */
  }
}

Figure 4.3: An Example of an Event Handler Performing Non-Blocking I/O
/* Step 1: initialize the control block */
aiocb.aio_fd = socket;
aiocb.aio_buf = buffer;
aiocb.aio_nbytes = bytes_to_write;
...
/* Step 2: initiate the operation; returns immediately */
aio_write(&aiocb);
do {
  ...
  /* perform other activities */
  ...
  /* Step 3: poll for completion or fatal error */
  error = aio_error(&aiocb);
} while (error == EINPROGRESS);
if (error != 0) {
  /* handle fatal error */
}
/* Step 4: obtain the return value, in this case,
* the number of bytes written */
return_value = aio_return(&aiocb);

Figure 4.4: An Example of Asynchronous I/O
Chapter 5

Servers Studied

In this chapter, I focus on the two servers, thttpd [7] and Flash [6], that I use in my evaluation. In particular, I describe their respective architectures and the changes to them that I implemented for my evaluation.

5.1 thttpd

thttpd has an event driven architecture. It has an event loop which polls for events and invokes the associated event handlers. All sockets are configured in non-blocking mode. An event is received in the event loop when a socket becomes ready for a read() or a write(), and the corresponding event handler is invoked. thttpd makes calls to blocking operations like open(), stat(), and sendfile() from within its event handlers. thttpd stalls if these operations block on disk I/O.

I modified thttpd to use the LAIO API. I call this version of thttpd as thttpd-LAIO. In this version all blocking system calls are invoked via laio_syscall(). Continuation functions are defined to handle laio_syscalls that block and finish asynchronously. I compare thttpd-LAIO with a regular thttpd just termed as thttpd.

5.2 Flash

Flash employs the asymmetric multiprocess event driven (AMPED) architecture [6]. It has an event driven core to handle all non-blocking operations. Additionally, it has helper
threads to handle all blocking operations. In Flash all sockets are used in non-blocking mode. All non-blocking socket operations are handled by the event driven core, and all potentially blocking operations like file open or read are handled by the helper threads. The thread running the event driven core dispatches work to the helper threads via remote procedure call (RPC) mechanism. Likewise, the helper threads employ RPC to notify the main thread of the completion of their tasks.

The original version of Flash mapped files in memory and subsequently wrote them to sockets [6]. After accepting a connection and reading the requested URL, Flash opens the requested file and maps it to its address space by the `mmap`() system call. Since opening a file is a potential blocking operation it is done by a helper thread. After the file is mapped it is written to the corresponding connection by the `writev()` system call. Although the socket for the connection is non-blocking, the writev system call could still block if the memory mapped file is not resident in memory, and the thread would incur a page fault in that case. Since memory residency of files is not guaranteed and page faults are not acceptable in the server core thread, Flash employed the `mmap()` system call (on FreeBSD) to check for memory residency of concerned pages. If pages were found to be not in memory, explicit I/O was issued by helper threads; I refer to these helper threads as read helper threads. This is how blocking due to page faults was avoided. I call this version of Flash as Flash-AMPED-mmap.

Current version of Flash supports the `sendfile()` system call. In this version the event driven core reads the requested URL after accepting the connection. The corresponding file is opened by a helper thread. Then the event driven core uses sendfile to write the file to the corresponding socket. Although the socket is used in non-blocking mode, sendfile can block if the file data is not in memory, causing the server thread to block. This is avoided by using an optimized `sendfile()` [9]. This requires a patch in the kernel.
which prevents sendfile( ) from blocking on disk I/O and returns a special errno.

The event core catches this errno and issues explicit I/O via read helper threads so that the
file data is brought in memory. I call this version of Flash as Flash-AMPED-sendfile. Note,
this version of Flash does I/O in a lazy manner which is analogous to LAIO. It calls send-
file( ) expecting it not to block, but if it blocks on disk I/O, the sendfile call returns with
a special error code, which is used to initiate I/O via helper threads. As a result, the server
main thread does not block. As such I do not expect the corresponding LAIO version of
Flash to outperform Flash-AMPED-sendfile but to match it.

I have modified both Flash-AMPED-mmap and Flash-AMPED-sendfile to get the cor-
responding event driven versions. I call these versions Flash-Event-mmap and Flash-Event-
sendfile respectively. In either version, there are no helper threads, instead, all helper func-
tions are called directly by the main thread. This means, for operations like file open, the
server may block, as would happen in a pure event driven server. For Flash-Event-mmap I
do not use the mincore operation. This is because, even if mincore detects that some page
is not in memory, the main thread would have to block at the I/O for the page. In such
cases the mincore operation is nothing but an overhead. Instead, I allow writev( ) to
incur page faults when pages are not in memory. For Flash-Event-sendfile I do not use the
optimized sendfile( ) because of similar reason.

I have modified both Flash-Event-mmap and Flash-Event-sendfile to use the LAIO li-
brary to prevent any blockings. I call these versions Flash-LAIO-mmap and Flash-LAIO-
sendfile respectively. In either version, all potentially blocking operations are identified
and the LAIO API is used to prevent (possible) blockings, for example, the writev( ) or
sendfile( ) system calls. Since LAIO is a universal API to handle any potentially block-
ing operation asynchronously, I do not use non-blocking sockets in Flash-LAIO-mmap and
Flash-LAIO-sendfile. Instead, all sockets are synchronous and I let the LAIO API to take
care of any blocking events that may happen. In Flash-LAI mmap I do not use mincore(), because, even if pages are not in memory for a writev(), the server does not block because of the LAIO API. Similarly, for Flash-LAI0-sendfile I do not use the optimized sendfile().
Chapter 6

Evaluation Methodology

In this chapter I present the methodology I used to evaluate the LAIO library. I used two web server applications for this purpose - thttpd and Flash. As described earlier I have two different versions of thttpd, viz., thttpd-libevent and thttpd-LAIO, and six different versions of Flash, viz., Flash-Event-mmap, Flash-Event-sendfile, Flash-AMPED-mmap, Flash-AMPED-sendfile, Flash-LAIO-mmap, and Flash-LAIO-sendfile. I compare the performance of thttpd-Event and thttpd-LAIO, similarly I compare the three versions of Flash using mmap and the other three versions of Flash using sendfile.

I used a couple of trace based web workload for this purpose. These workloads are obtained from the academic web servers at Rice University (Rice workload) and the University of California at Berkeley (Berkeley workload). Use of these workloads exists in published literature [15]. Each workload is associated with a trace file of web requests.

<table>
<thead>
<tr>
<th>Web Workload</th>
<th>No. of requests</th>
<th>Small ($\leq 8$ KB)</th>
<th>Medium ($&gt; 8$ KB and $\leq 256$ KB)</th>
<th>Large ($&gt; 256$ KB)</th>
<th>Total footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rice</td>
<td>245,820</td>
<td>5.5%</td>
<td>20.2%</td>
<td>74.3%</td>
<td>1.1 Gigabytes</td>
</tr>
<tr>
<td>Berkeley</td>
<td>3,184,540</td>
<td>8.2%</td>
<td>33.2%</td>
<td>58.6%</td>
<td>6.4 Gigabytes</td>
</tr>
</tbody>
</table>

Table 6.1 : Web Trace Characteristics

Table 6.1 shows the characteristics of the Rice and Berkeley web workloads. The total
number of requests in the Rice and the Berkeley workload are 245,820 and 3,184,540 respectively. Off the total bytes transferred, the fractions contributed by small files (size less than or equal to 8 Kilobytes), medium files (size in between 8 Kilobytes and 256 Kilobytes), and large files (size greater than 256 Kilobytes) are shown in columns Small, Medium, and Large respectively. Total footprint for the Rice and the Berkeley workloads are 1.1 Gigabytes and 6.4 Gigabytes respectively.

The trace file associated with each workload is a set of request sequences. A sequence consist of one or more requests. Each sequence begins with a connection set up and ends with a connection tear down. Requests in a sequence are sent one at a time, the response is read completely, and then the next request is sent. All requests in a sequence are sent over a persistent HTTP connection. If the server does not support persistent connections, like httpd, each request is sent over a non-persistent HTTP connection, that is, the connection is set up and torn down before and after each request respectively.

I used a program that simulates concurrent clients sending web requests to a web server. The number of concurrent clients can be varied with this program. It supports persistent and non-persistent connections. The program simulates multiple clients that play the request sequences in the trace against the server. The program terminates when the trace is exhausted and reports overall throughput and response time.

I used a Pentium Xeon 2.4 GHz machine with 2 Gigabytes of memory as my server machine. It runs FreeBSD-5 which supports KSE, FreeBSD’s scheduler activation implementation. An identical machine was used as the client machine. The server and client machines were connected by a Gigabit Ethernet switch.
Chapter 7

Results

In this chapter I present experimental results obtained. Section 7.1 shows some micro-
benchmark results. Section 7.2 covers the thttpd experiments. Section 7.3 covers the Flash
experiments.

7.1 Microbenchmarks: LAIO vs. non-blocking I/O vs. POSIX AIO

In order to compare the cost of performing I/O using LAIO, non-blocking I/O, and POSIX
AIO, I implemented a set of microbenchmarks. Specifically, these microbenchmarks mea-
sured the cost of 100,000 iterations of reading a single byte from a pipe under various cir-
cumstances. For POSIX AIO, the microbenchmarks include calls to aio_error() and
aio_return() in order to obtain the read’s error and return values, respectively. I used a
pipe so that irrelevant factors, such as disk access latency, did not affect my measurements.
Furthermore, the low overhead of I/O through pipes would emphasize the differences be-
tween the three mechanisms. In one case, when the read occurs a byte is already present in
the pipe, ready to be read. In the other case, the byte is not written into the pipe until the
reader has performed either the LAIO or the aio_read(). In this case, I did not measure
the cost of a non-blocking read because the read would immediately return EAGAIN.

As would be expected, when the byte is already present in the pipe before the read, non-
blocking I/O performed the best. LAIO was a factor of 1.4 slower than non-blocking I/O;
and AIO was a factor of 4.48 and 3.2 slower than non-blocking I/O and LAIO, respectively.
In the other case, when the byte was not present in the pipe before the read, I found that LAIO was a factor of 1.08 slower than AIO.

In these microbenchmarks, only a single byte was read at a time. Increasing the number of bytes read at a time, did not change the ordering among LAIO, non-blocking I/O, and POSIX AIO as to which performed best.

In addition, I observed that enabling scheduler activations within an address space resulted in a small added overhead to all system calls by any thread within that same address space. To characterize this overhead, I measured the cost of 1,000,000 iterations of calling getpid(). I found that enabling scheduler activations made getpid() a factor of 1.05 slower. In effect, this represents an indirect cost to users of LAIO or asynchronous I/O implementations using a pool of threads.

7.2 thttpd Results

I subjected the thttpd server to traces collected from real web servers to obtain the following results. In Figure 7.1 I see the throughput of the cold and warm cache cases for Berkeley CS department workload. thttpd-LAIO achieves more than 38% higher throughput than thttpd, because the workload is too big to fit in memory, thus system calls like sendfile(), stat() and open() block on disk reads often. This blocks the thttpd server which is not the case for thttpd-LAIO. For the warm cache case, we note there isn't much difference, this is because compulsory misses are not the dominating factor.

Figure 7.2 shows the response time for the Berkeley trace for cases of cold and warm caches. We notice that under heavy load there is a significant improvement of more than 30% in the response time for thttpd-LAIO.

Figures 7.3 shows the throughput for the Rice CS department workload for cold and warm caches. Unlike the Berkeley workload this workload fits in memory. So for the cold
Figure 7.1: Throughput for Berkeley workload with thttpd

Figure 7.2: Response time for Berkeley workload with thttpd
Figure 7.3: Throughput for Rice workload with thttpd

Figure 7.4: Response time for Rice workload with thttpd
Figure 7.5: Throughput for Rice workload with thttpd with reduced memory size

cache we see some gain from LAIO, this is basically due to compulsory misses, causing the server to make disk reads. For the warm cache we don’t see any gain from LAIO as all the files are already in memory and there is no blocking on I/O. We actually see a little performance degradation (less than 2%).

Figure 7.4 shows the response time for the Rice trace for cases of cold and warm caches. For the warm case we find the thttpd-LAIO follows thttpd closely. For the cold cache case thttpd-LAIO has a lower response time, again this is due compulsory misses.

To verify the correlation between the memory size, the workload and performance, I repeated the experiments with the Rice trace, but decreased the physical memory to 512MB. Figure 7.5 shows the throughput for the Rice workload for cold and warm caches on a server having 512MB of RAM. This workload, using this setup, behaves very similar to the Berkeley workload as it doesn’t fit into the server’s physical memory.
Figure 7.6: Response time for Rice workload with thttpd with reduced memory size.

Figure 7.6 shows the response time for the Rice trace for cases of cold and warm caches. Again behaves very similar to the Berkeley workload.

### 7.3 Flash Results

In this section I compare the performance of different versions of Flash, as described in Chapter 5.2, for the Rice workload and the Berkeley workload. 500 concurrent clients were used to send the workload requests to the server. I measured the overall throughput reported by the client program. For each server version I do one cold cache run followed immediately by a warm cache run, for each workload.

I also conducted a series of experiments where I varied the number of clients, as with the thttpd experiments. The results for the various versions of Flash conform to my expectations. In particular, the results scale similarly to the thttpd results as described in
Section 7.2. The relative ordering of the various versions of the flash remain the same as described in the next two subsections. Hence, I argue that the results with 500 clients, for the various versions of Flash, are representative of the overall experiments.

7.3.1 mmap Results

First, I compare the performance of Flash-Event-mmap, Flash-AMPED-mmap and Flash-LAIO-mmap. All these versions use `mmap()` to map the requested file to memory, and `writev()` to send the file across the connection. Additionally, Flash-AMPED-mmap uses the `mlock()` system call to check for memory residency of pages before sending them out (as described in Chapter 5.2).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Flash-Event-mmap</th>
<th>Flash-AMPED-mmap</th>
<th>Flash-LAIO-mmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold cache</td>
<td>203.16 Mbps</td>
<td>385.83 Mbps</td>
<td>298.76 Mbps</td>
</tr>
<tr>
<td>Warm cache</td>
<td>830.22 Mbps</td>
<td>799.74 Mbps</td>
<td>796.82 Mbps</td>
</tr>
</tbody>
</table>

Table 7.1: Mmap results: Throughput for the Rice workload with 500 clients

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Flash-Event-mmap</th>
<th>Flash-AMPED-mmap</th>
<th>Flash-LAIO-mmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold cache</td>
<td>81.18 Mbps</td>
<td>133.59 Mbps</td>
<td>132.02 Mbps</td>
</tr>
<tr>
<td>Warm cache</td>
<td>78.36 Mbps</td>
<td>126.67 Mbps</td>
<td>131.21 Mbps</td>
</tr>
</tbody>
</table>

Table 7.2: Mmap results: Throughput for the Berkeley trace with 500 clients

Table 7.1 and Table 7.2 show the throughput of the three server configurations for the Rice workload and Berkeley workload with 500 concurrent clients respectively. Results for
a cold cache run followed by a warm cache run are shown. Recall, that the total footprint of
the Rice workload is 1.1 Gigabytes and that of the Berkeley workload is 6.4 Gigabytes. The
server machine had 2 Gigabytes of memory, so in the warm cache run the entire workload
fits in memory for the Rice workload. But, for the Berkeley workload, there is considerable
I/O even for the warm cache run.

For both the workloads in cold cache runs, the throughput attained by Flash-AMPED-
mmap is higher than that of Flash-Event-mmap. Specifically, for the Rice workload Flash-
AMPED-mmap has about 90% performance improvement over Flash-Event-mmap, while,
for the Berkeley workload it has about 65% performance improvement. Flash-Event-mmap
blocks on operations like open(), stat(), and writev(), and hence the performance
difference. For cold cache runs in both workloads Flash-LAIO-mmap betters Flash-Event-
mmap because of the same reasons stated above. For warm cache runs in both the work-
loads Flash-LAIO-mmap closely matches Flash-AMPED-inmap.

For the Rice workload in warm cache run, Flash-Event-mmap outperforms Flash-AMPED-
mmap by about 4%. For this workload in warm cache run the entire workload fits in
memory. In this case, the mincore operations of Flash-AMPED-mmap is nothing but an
overhead, and hence the performance difference.

For the Rice workload in cold cache run, Flash-AMPED-mmap outperforms Flash-
LAIO-mmap by about 29%. I measured the number of time Flash-AMPED-mmap calls
the read helper, which does I/O on behalf of the server main thread. This number was
41072. Flash-LAIO-mmap does not have any helper thread, instead it page faults on the
writev() system call if the data is not memory. The number of page faults was 46486, which
is about 13% higher than the number of calls to the read helper in Flash-AMPED-mmap.
Note, that the purpose of the read helper and the page faults is to perform I/O and bring the
required data in memory. So, the same amount of I/O is being performed in both cases but
it takes 13% more I/O operations in Flash-LAIO-mmap. The difference in the prefetching algorithm of the file system prefetch of read() system call and the virtual memory system prefetch of page faults is causing the performance difference here. In particular, file system prefetch is more aggressive than virtual memory system prefetch.

For the Berkeley workload in cold cache run the number of calls to the read helper in Flash-AMPED-mmap was 689167, while the number of page faults in Flash-LAIO-mmap was 690835. For the warm cache run, these numbers were 681320 and 670782. These explain why Flash-AMPED-mmap performs a little better than Flash-LAIO-mmap in cold cache case and vice versa in the warm cache case.

For the Berkeley workload in cold cache run there is considerable blocking in Flash-Event-mmap because the workload does not fit in memory. Hence, Flash-Event-mmap performs worse than the other two servers.

### 7.3.2 sendfile Results

In this subsection I compare the versions of Flash that use sendfile() to transfer data across the network. sendfile() writes file referenced by an opened file descriptor to a socket. It can block if the data for the file is not resident. However, Flash-AMPED-sendfile utilizes an optimized version of sendfile that sends a special errno on blocking on disk, and the server can continue its operation without blocking. The server main thread then does an I/O via the read helper thread to bring the file data in memory. In my experiments Flash-AMPED uses this optimized sendfile, while Flash-Event and Flash-LAIO use the normal sendfile. A kernel patch from the original developers of Flash is required to get the optimized sendfile; this is described in Chapter 5.2. Recall that we do not expect Flash-LAIO-sendfile to outperform Flash-AMPED-sendfile, but to match its performance.

Table 7.3 and Table 7.4 show the results for the Rice workload and the Berkeley work-
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Flash-Event-sendfile</th>
<th>Flash-AMPED-sendfile</th>
<th>Flash-LAIO-sendfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold cache</td>
<td>276.87 Mbps</td>
<td>397.78 Mbps</td>
<td>381.54 Mbps</td>
</tr>
<tr>
<td>Warm cache</td>
<td>844.76 Mbps</td>
<td>843.45 Mbps</td>
<td>815.49 Mbps</td>
</tr>
</tbody>
</table>

Table 7.3: Sendfile results: Throughput for the Rice workload with 500 clients

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Flash-Event-sendfile</th>
<th>Flash-AMPED-sendfile</th>
<th>Flash-LAIO-sendfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold cache</td>
<td>121.59 Mbps</td>
<td>170.59 Mbps</td>
<td>171.43 Mbps</td>
</tr>
<tr>
<td>Warm cache</td>
<td>124.60 Mbps</td>
<td>180.42 Mbps</td>
<td>178.89 Mbps</td>
</tr>
</tbody>
</table>

Table 7.4: Sendfile results: Throughput for the Berkeley trace with 500 clients

Load under cold cache and warm cache runs respectively.

For both the workloads under cold cache run Flash-Event-sendfile performs worse than the other two servers. This is because it blocks at file `open()`, `stat()`, and `sendfile()` while the other two servers do not block under such conditions.

For Rice workload in warm cache run, Flash-Event-sendfile matches the performance of Flash-AMPED-sendfile. This is because the entire workload is cached and fits in memory, so the calls to `sendfile()` do not block for disk I/O.

For Berkeley workload in warm cache run, however, Flash-Event-sendfile performs worse than Flash-AMPED-sendfile. This is because the workload does not fit in memory and Flash-Event-sendfile blocks on disk I/O.

For Rice workload in cold cache run, Flash-LAIO-sendfile closely matches the performance of Flash-AMPED-sendfile. For Berkeley workload in both cold cache run and warm cache run, Flash-LAIO-sendfile matches the performance of Flash-AMPED-sendfile. This is because the optimized `sendfile()` does I/O lazily like LAIO.
For Rice workload in warm cache run, Flash-LAIO-sendfile performs about 3% worse than Flash-AMP-sendfile. This is because of the cost of LAIO as explained in Section 7.1.
Chapter 8

Related Work

Some prior work has been done in this area. Other than AIO, Windows NT [3] and VAX/VMS [1] operating systems have provided asynchronous I/O.

In Windows NT, the application can start an I/O operation then do other work while the device completes the operation. When the device finishes transferring the data, it interrupts the application’s calling thread and copies the result to its address space. The kernel uses a Windows NT asynchronous notification mechanism called asynchronous procedure call (APC) to notify the application’s thread with the completion of the I/O operation.

Like NT, VAX/VMS allows for a process to request that it gets interrupted when an event occurs, such as an I/O completion event. The interrupt mechanism used is called asynchronous system trap (AST) which provides a transfer of control to user-specific routine that handles the event.

Similar to AIO, asynchronous notifications in both VAX/VMS and Windows NT are limited to few events, mainly I/O operations and timers. This is not broad enough to support any system call like in I AIO. Also, asynchronous I/O in both Windows NT and VAX/VMS is not lazy.
Chapter 9

Conclusions

In this thesis I have introduced Lazy Asynchronous I/O (LAIO), a new API for performing I/O that befits event-driven servers. I have pointed out the shortcomings of both asynchronous and non-blocking I/O with respect to server performance. I have explained the design and implementation of LAIO, demonstrated that it subdues the shortcomings of the earlier APIs. Using a micro-benchmark, LAIO was shown to be more than 3 times faster than AIO when the data was already available in memory. It also had a comparable performance to AIO when actual I/O needed to be made. I have also shown that thttpd achieved more than 38% increase in its throughput using LAIO. The Flash web server’s throughput, originally achieved with kernel modifications, was matched using LAIO without making kernel modifications.

As I have shown how LAIO can be used with event notification libraries or directly with event driven servers, it can also be used with user threading libraries. User threading libraries like Capriccio [12] running on a single kernel thread per CPU can’t afford to block as this would block the whole process. Now, they can uniformly use LAIO for all their blocking operations.
Bibliography


[8] Niels Provos. Libevent - an event notification library. Version 0.7c is available from the author’s web site, http://www.monkey.org/~provos/libevent/, October 2003. Libevent is also included in recent releases of the NetBSD and OpenBSD operating systems.


